Modèles de Programmation

Emmanuel Chailloux

Informations sur le cours MdP

Site

`http://www-apr.lip6.fr/~chaillou/Public/enseignement/2013-2014/MdP/`

Email

`Emmanuel.Chailloux@lip6.fr`

Description de l'UE

- Comprendre les différents modèles de programmation séquentielle (impératif, fonctionnel, modulaire, objet)
- Maîtriser leur mise en pratique dans le cadre du langage OCaml,
- Apprécier la sûreté et la réutilisation apportéee par le typage statique
- Savoir utiliser un environnement de développement pour Caml
- Comprendre les différents modèles de programmation concurrente en fonction des modèles mémoire utilisés (mémoire partagée, mémoire répartie).

mots clés :

Panorama des langages de programmation, Types et déclarations, Mécanismes d'abstraction, Programmation par objets, Programmation fonctionnelle, Programmation modulaire, Systèmes de types, Programmation concurrente

Plan du cours (1)

1. présentation de l'UE
   généralités sur les langages et les modèles de programmation,
   le noyau fonctionnel Caml : définitions et appels de fonction, valeurs fonctionnelles
2. le noyau fonctionnel : déclarations de types, filtrage de motifs, polymorphisme
   Exceptions : déclenchement, récupération et style de programmation
3. Traits impératifs : valeurs physiquement modifiables et structures de contrôle, variables de type faibles
4. Modules simples et paramétrés : séparation interface et implantation, compilation séparée, types abstraits de données
5. Objet I : classes, instances, héritage, liaison tardive
   Objet II : sous-typage et polymorphisme objet
6. Concurrence et répartition

Logiciels du cours

indépendants des systèmes (Windows, Linux, Macosx, ...) :

- langage Objective Caml 4.01
  - pré-installé sur les machines de l'école doctorale (à vériifer)
  - à installer à la maison à partir du site de l'Inria :
    `http://caml.inria.fr/`
- Environnements de développement
  - Emacs et mode Tuareg
    `http://www.gnu.org/software/emacs/` et
    `http://tuareg.forge.ocamlcore.org/`
  - Eclipse et plug-in Ocaide
    `http///www.eclipse.org` et `http://www.algo-prog.info/ocaide/`

Bibliographie

- Pascal Manourry. Programmation de droite à gauche et vice-versa, paracamplus, 2012.
- Xavier Leroy et al. The Objective Caml system : documentation and user's manual Inria, 2010.
- Emmanuel Chailloux, Pascal Manoury et Bruno Pagano. Développement d'Applications avec Objective Caml. O'Reilly, 2000.
- Guy Cousineau et Michel Mauny. Approche fonctionnelle de la programmation. Dunod, 1995.
- Philippe Nardel. Programmation fonctionnelle, générique et objet : Une introduction avec le langage OCaml. Vuibert, 2005
- Yaron Minsky, ANul Madhavapeddy, Jason Hickey. Real World OCaml, O'Reilly 2013

autres références sur : `http://caml.inria.fr/about/books.en.html` et `scriptsizehttp://ocaml.org/`

Objective Caml

- langage fonctionnel,
- typé statiquement,
- polymorphe paramétrique,
- avec inférence de types,
- muni d'un mécanisme d'exceptions,
- et de traits impératifs
- possèdant un système de modules paramétrés
- et un modèle objet
- exécutant des processus légers
- et communiquant sur le réseau Internet,
- indépendant de l'architecture machine.

Historique

- l'ancêtre : ML (*meta-langage*) de LCF (80)
- machines abstraites (84)
  - la FAM : Cardelli
  - la CAM : Curien, Cousineau
- spécifications : Standard ML (84 - Milner)
- premières implantations
  - CAML - Suarez - Weis - Mauny (87)
  - SML/NJ - Mc Queen - Appel (ATT - 88)
- nouvelles implantations : Caml-light (90) Leroy - Doligez
- modules paramétrés : Caml Special Light (95)
- extension objet (96)
- labels, options, variants polymorphes, ...

Mise en œuvre (1)

- compilateur natif (commande **ocamlopt**)
  - pour Intel, Sparc, HP-pa, PowerPC, ...
  - pour Linux, Macosx, Windows, ...

```
$ cat t.ml
let f x = x + 1 ;;
print_int (f 18) ;;
print_newline () ;;

$ ocamlopt -o tn.exe t.ml
$ ./tn.exe
19
```

Mise en œuvre (2)

- compilateur byte-code (commande **ocamlc**)
  - ligne de commande

```
$ cat t.ml
let f x = x + 1 ;;
print_int (f 18) ;;
print_newline () ;;

$ ocamlc -i -custom -o tb.exe t.ml
val f : int -> int
$ ./tb.exe
19
```

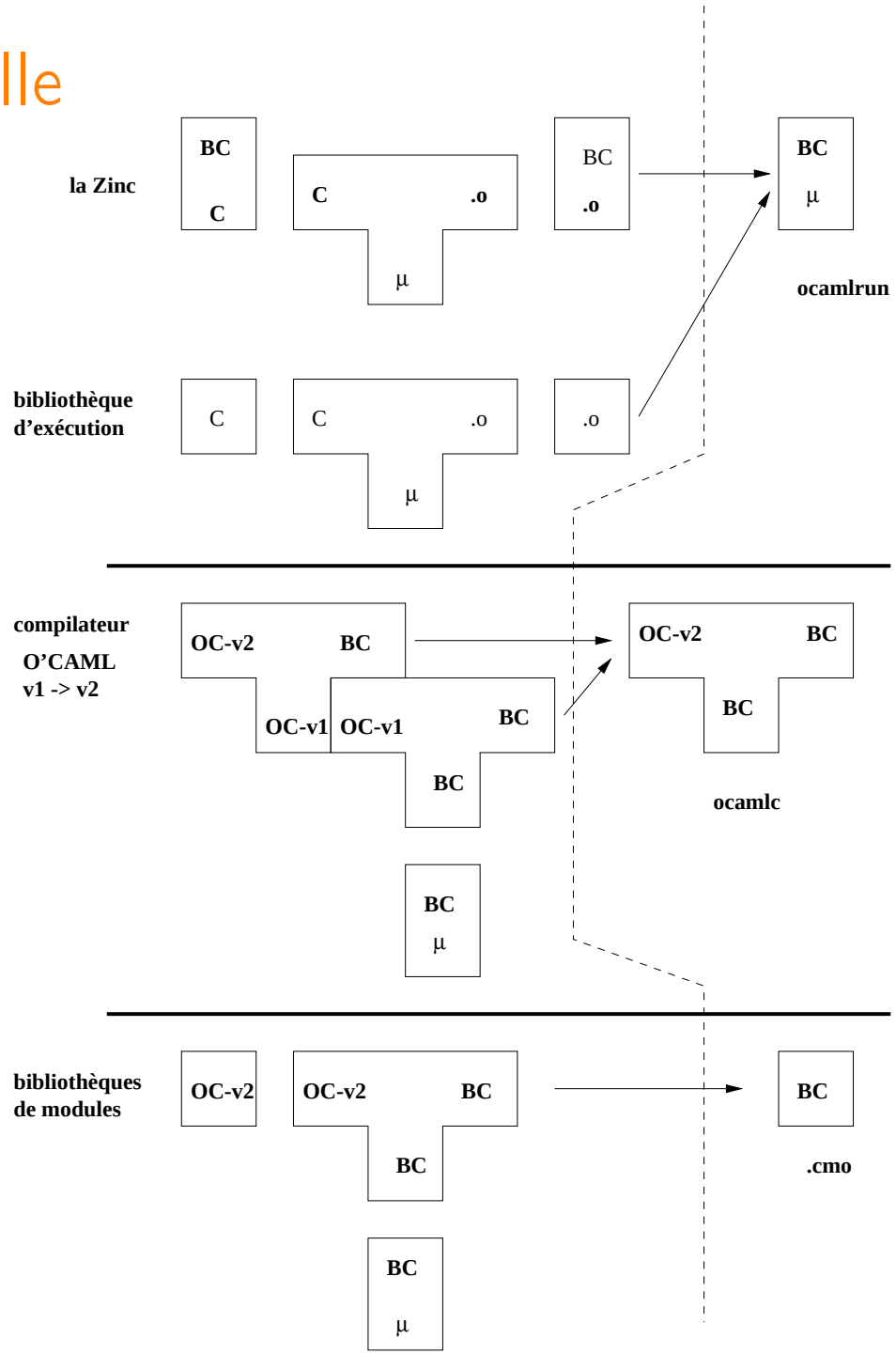
Mise en œuvre (3)

- compilateur byte-code (commande **ocaml**)
  - boucle d'interaction

```
1  $ ocaml
2           Objective Caml version 3.10.2
3
4  # let f x = x + 1 ;;
5  val f : int -> int = <fun>
6  # f 18 ;;
7  - : int = 19
8  #  #quit ;;
9  $
```

  - $: invite Unix
  - # : invite O'Caml
  - #quit : directive O'Caml

Machine virtuelle



Phrases du langage

Se terminent par `;;`

- expressions
- déclarations
  - de valeurs
  - de types
  - d'exceptions

- déclarations de modules
- déclarations de classes

Notation qualifiée `Module.nom` pour accéder à un champs d'un module.

Noyau fonctionnel

Programmes en Objective Caml

Seuls les programmes **correctement typés** peuvent être exécutés.

- calcul d'une expression
- application d'une fonction à ses arguments
- évaluation immédiate des arguments
- rupture d'évaluation si calcul impossible

Types de base, valeurs et fonctions

- nombres
  - *int* ($[-2^{30}, 2^{30}-1]$ (m32)
  - *float* (IEEE 754) mantisse 53bits, exposant $\in [-1022, 1023]$
- caractères : *char*
- chaînes de caractères : *string*
- booléens : *bool*

Opérations sur les nombres

entiers		flottants	
`+`	addition	`+.`	addition
`-`	soustraction	`-.`	soustraction
`*`	multiplication	`*.`	multiplication
`/`	division entière	`/.`	division
`mod`	modulo	`**`	exponentiation

opérateurs infixes!!!

```
1 # 1 + 3;;
2 - : int = 4
3 # 1.8 *. 9.1;;
4 - : float = 16.38
```

Fonctions sur les nombres

entiers		flottants	
`ceil`		`cos`	cosinus
`floor`		`sin`	sinus
`sqrt`	racine carrée	`tan`	tangente
`exp`	exponentielle	`acos`	arccosinus
`log`	log népérien	`asin`	arcsinus
`log10`	log en base 10	`atan`	arctangente

angles en radiant!!!

Calculs sur les nombres (1)

```
1  # 1 + 2;;
2  - : int = 3
3
4  # 9/2;;
5  - : int = 4
6
7  # 2147483650;;
8  - : int = 2
9
10 # 9.1 /. 2.2;;
11 - : float = 4.13636363636
12
13 # 1. /. 0.;;
14 - : float =           Inf
```

Calculs sur les nombres (2)

```
1  # 2 + 2.1;;
2  This expression has type float but
3  is here used with type int
```

Un *int* ne peut pas remplacer un *float*!!!

```
1  # sin;;
2  - : float -> float = <fun>
3
4  # asin 1.;;
5  - : float = 1.57079632679
```

Calculs sur les chaînes

```
1  # 'B';;
2  - : char = 'B'
3
4  # int_of_char 'B';;
5  - : int = 66
6
7  # "est une chaine";;
8  - : string = "est une chaine"
9
10 # (string_of_int 1987)^" est l'annee de CAML";;
11 - : string = "1987 est l'annee de CAML"
```

Opérations sur les booléns et relations

`not`	négation		
`&&`	et séquentiel	`&`	synonyme pour et
`||`	ou séquentiel	`or`	synonyme pour ou
`=`	égalité structurelle	`<`	inférieur
`==`	égalité physique	`>`	supérieur
`<>`	négation de =	`<=`	inférieur ou égal
`!=`	négation de ==	`>=`	supérieur ou égal

égalités et inégalités polymorphes!!!

```
= : 'a -> 'a -> bool
== : 'a -> 'a -> bool
```

Produits cartésiens, n-uplets

- constructeur de valeurs : `,`
- constructeur de types : `*`
- accesseurs (couples) : `fst` et `snd`

```
1  # (18,"mars");;
2  - : int * string
3  # (18,"mars",1999);;
4  - : int * string * int
5  # fst (18,"mars") ;;
6  - : int = 18
7  # snd (18,"mars",1999)
8    erreur de typage
```

Listes homogènes

- liste vide : `[]`
- constructeur `::`
- type *list*
- accesseurs `List.hd` et `List.tl`

```
1  # [];;
2  - : 'a list
3  # 1::2::3::[];;
4  - : int list
5  # [1; 2; 3;];;
6  - : int list
7  # [1; "hello"; 3];;
8  erreur de typage
9  # List.hd [1.1; 1.2; 1.3];;
10 - : float = 1.1
11 # List.hd [];;
12 exception non récupérée
```

Expression conditionnelle

Syntaxe:

if expr1 **then** expr2 **else** expr3

- *expr1* de type *bool*
- *expr2* et *expr3* de même type

```
1  # if 3 = 4 then 0 else 4;;
2  - : int = 4
3  # if 5 = 6 then 1 else "Non";;
4  - : erreur de typage
5  # (if 3 = 5 then 8 else 10) + 5;;
6  - : int = 15
7  # 5 = 6 || 7 = 9;;
8  - : bool = false
```

Déclarations de valeurs (1)

Déclarations globales:

Syntaxe:

let p = e

```
1 # let pi = 3.14159;;
2 val pi : float = 3.14159
3
4 # sin (pi /. 2.0);;
5 - : float = 0.999999999999
```

Déclarations de valeurs (2)

Déclatations locales:

Syntaxe:

let p = e1 **in** e2

```
1  # let x = 3 in
2      let b = x < 10 in
3        if b then 0 else 10;;
4  - : int = 0
5
6  # b;;
7  Unbound value b
```

Déclarations combinées

Syntaxe:

let $p_1 = e_1$ **and** $p_2 = e_2$ ... **and** $p_n = e_n$ **;;**

```
1 # let x = 1 and y = 2;;
2 val x : int = 1
3 val y : int = 2
4 # x+y;;
5 - : int = 3
```

Syntaxe:

let $p_1 = e_1$ **and** $p_2 = e_2$ ... **and** $p_n = e_n$ **in** expr **;;**

```
1 # let a = 3.0 and b = 4.0 in sqrt(a*.a+.b*.b);;
2 - : float = 5
```

Valeurs fonctionnelles, fonctions

Syntaxe:

function p -> e

- fonction anonyme : **function** p -> e
- de type : typede(p) -> typede(e)

```
1  # function x -> x+1;;
2   - : int -> int
3  # (function x -> x+1) 1999;;
4  - : int = 2000
5  # function x -> if x < 0 then -x else x;;
6  - : int -> int
7  # function x -> (function y -> 2*x+3*y);;
8  - : int -> int -> int
```

Déclaration de valeurs fonctionnelles

```
# let succ = function x -> x+1;;
succ : int -> int
# succ 420;;
- : int = 421
```

Syntaxe:

let f = **function** p -> e *ou* **let** f p = e

ou **let** f (p) = e

```
# let pred (x) = x -1;;
pred : int -> int = <fun>
# let f c = 2*(fst c) + 3*(snd c);;
f : int * int -> int = <fun>
# f (1,2);;
- : int = 8
# let g = function x -> (function y -> 2*x + 3*y);;
g : int -> int -> int = <fun>
# g 1 2;;
- : int = 8
```

Portée des déclarations

- portée lexicale (liaison statique)

```
1  # let p = q + 1;;          erreur q inconnue
2
3  # let p = p + 1;;          erreur p inconnue
4
5  # let p = 10;;             p : int = 10
6
7  # let addp x = x + p;;     addp : int ->int = <fun>
8
9  # addp 8;;                 - : int = 18
10
11 # let p = 40;;             p : int = 40
12
13 # addp 8;;                 - : int = 18
14
15 # p;;                      - : int = 40
```

Récursivité

Syntaxe:

let rec p = *expr*

```
1  # let rec sigma x = if x = 0 then 0
2                      else x + sigma(x-1);;
3  sigma : int -> int = <fun>
4
5  # sigma 10;;
6   - : int = 55
```

récursion mutuelle:

```
1  # let rec odd x = if x = 0 then false else even(x-1)
2  # and even x = if x = 0 then true else odd(x-1);;
3  val odd : int -> bool = <fun>
4  val even : int -> bool = <fun>
5
6  # odd 27;;
7  - : bool = true
```

Polymorphisme paramétrique (1)

- même code pour des arguments de types différents
- la fonction n'utilise pas la structure de l'argument

```
1  # let mp a b = a,b;;
2  val mp : 'a -> 'b -> 'a * 'b = <fun>
3
4  # let x = mp 3 6.8;;
5  val x : int * float = 3, 6.8
6
7  # let y = mp true [1];;
8  val y : bool * int list = true, [1]
9
10 # fst x;;
11 - : int = 3
```

Polymorphisme paramétrique (2)

- même code pour des arguments de types différents
- la fonction n'utilise pas la structure de l'argument

```
1  # let id x = x;;
2  val id : 'a -> 'a = <fun>
3
4  # let app x y = x y;;
5  val app : ('a -> 'b) -> 'a -> 'b = <fun>
6
7  # app id 1;;
8  - : int = 1
```

Résumé des expressions rencontrées

```
expr ::= constante
       | ( expr )
       | ident
       | Mod.ident
       | op expr
       | expr infix-op expr
       | if expr then expr else expr
       | function ident -> expr
       | expr expr
       | expr , expr | epxr :: epxr
       | let [rec] ident = expr
           {and ident = expr} in expr
```

Exemples (1)

Comptage des éléments d'une liste:

```
1  # let rec long l =
2        if l = [] then 0
3        else 1 + long (List.tl l);;
4  val long : 'a list -> int = <fun>
5
6  # long [2;4;7];;
7  - : int = 3
8
9  # long [];;
10 - : int = 0
11
12 # long ['A'; 'G'];;
13 - : int = 2
```

Exemples (2)

Composition de fonctions:

```
1  # let compose f g x = f (g x);;
2  val compose : ('a -> 'b) -> ('c -> 'a) ->
3                           'c -> 'b = <fun>
4
5  # let add2 x = x + 2 and mult5 x = x * 5;;
6  val add2 : int -> int = <fun>
7  val mult5 : int -> int = <fun>
8
9  # compose mult5 add2 9;;
10 - : int = 55
```

Exemples (3)

Application d'une fonction sur les éléments d'une liste:

```
1  # let rec map f l =
2    if l = [] then []
3    else
4      let t = List.hd l
5      and q = List.tl l in
6        (f t) :: (map f q) ;;
7  val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
8
9  # let l1 = map add2 [7; 2; 9];;
10 val l1 : int list = [9; 4; 11]
11
12 # let l2 = map long [['z';'a';'m']; ['.';'n';'e';'t']];;
13 val l2 : int list = [3; 4]
```

Filtrage de motif

permet l'accès aux structures de données

- en testant une valeur
- en nommant une partie de la structure

motif:

- assemblage correct (syntaxe et type) d'objets
  - de types de base (*int*, *bool*,...)
  - de paires, listes et constructeurs
  - d'identificateurs
  - du motif "sauvage" (_)
- ce n'est pas une expression (pas de calcul)

Syntaxe du filtage de motif

Syntaxe:

match e **with** p_1 -> e_1 | p_2 -> e_2 ... | p_n -> e_n

- filtrage séquentiel de e par les différents p_i.
- Si un des motifs correspond à la valeur de e, alors sa branche e_i est évaluée.
- tous les e_i du même type, idem pout les p_i
- motif linéaire ((x,x) interdit)
- détection d'un filtrage exhaustif

Exemple : fonction imply

```
1  # let imply v = match v with
2            (true,true) -> true
3          | (true,false)-> false
4          | (false,true)-> true
5          | (false,false)->true;;
6  val imply : bool * bool -> bool = <fun>
```

Autre version:

```
1  # let imply v = match v with
2            (true,x)  -> x
3          | (false,_) -> true;;
4  val imply : bool * bool -> bool = <fun>
```

Warnings

- filtrage non exhaustif :

```
# let f x = match x with
    (true, x) -> true
  | (false, false) -> true;;
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
(false, true)
val f : bool * bool -> bool = <fun>
```

- cas inutile :

```
# let f x = match x with
    (a,b) -> true
  | (true,false) -> false;;
Warning: this match case is unused.
val f : bool * bool -> bool = <fun>
```

Style Caml : définition par cas

en utilisant le filtrage de motifs sur les différents constructeurs d'uns structure de données :

```
1 let rec long l = match l with
2     [] -> 0
3 | _::q -> 1 + long q;;
4 val long : 'a list -> int = <fun>
```

```
1 let rec map f l = match l with
2   [] -> []
3 | t::q -> (f t) :: map f q;;
4 val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

Filtrage avec gardes

- expression conditionnelle après le motif
- reprise du filtrage si *false*

Syntaxe:

match e **with** p_1 **when** c_1 -> e_1 | p_2 -> e_2 ...| p_n -> e_n

---

```
1  # let rec mem_assoc3 x l = match l with
2        [] -> false
3    | (y1,y2,y3)::t when y1 = x -> true
4    |  _::t -> mem_assoc3 x t;;
5  val mem_assoc3 :
6    'a -> ('a * 'b * 'c) list -> bool = <fun>
```

Motif dans les déclarations

- **Formes équivalentes:**

  Syntaxe:

  match *e* **with** filtrage $\equiv$ (**function** filtrage) *e*

- **Déclarations déstructurantes:**

  Syntaxe:

  let $p = e$ p est un motif

```
1 # let (x,y) = (3,true);;
2 val x : int = 3
3 val y : bool = true
```